(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) Publication number: **0 652 518 A1**

(12) EUROPEAN PATENT APPLICATION

(21) Application number: **94117366.8**

(51) Int. Cl.[6]: **G06F 11/34**

(22) Date of filing: **03.11.94**

(30) Priority: **04.11.93 US 147645**

(43) Date of publication of application:
10.05.95 Bulletin 95/19

(84) Designated Contracting States:
DE FR GB

(71) Applicant: **MICROSOFT CORPORATION**
One Microsoft Way
Redmond,
Washington 98052-6399 (US)

(72) Inventor: **Bolosky, William J.**
24622 S.E. Mirrormont Drive
Issaquah,
Washington 98027 (US)
Inventor: **Rashid, Richard F.**
18601 N. E. 133rd Street
Woodinville,
Washington 98072 (US)

(74) Representative: **Patentanwälte Grünecker,**
Kinkeldey, Stockmair & Partner
Maximilianstrasse 58
D-80538 München (DE)

(54) Operating system based performance monitoring of programs.

(57) An operating provides a facility within its kernel for monitoring program performance. The facility may monitor user level programs as well as portions of the operating system, such as the kernel. The facility counts instructions and/or function calls to provide a useful performance metric to a user of the system. The count is forwarded to a user level monitoring program. The inclusion of the facility within the kernel enhances the speed of performance monitoring and enables the operating system to be directly monitored by the facility.

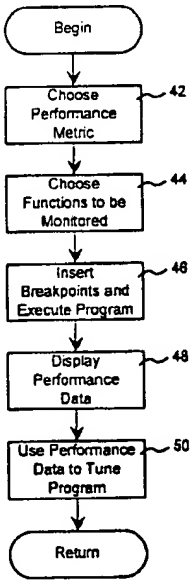

Figure 4

EP 0 652 518 A1

Technical Field

The present invention relates generally to data processing systems and, more particularly, to performance monitoring of programs run on data processing systems.

Background of the Invention

A number of different statistical techniques have been developed for monitoring performance of a program run on a data processing system. One of the most prominent techniques is to monitor the time spent in executing respective functions of the program. One difficulty with this approach is that in many instances the granularity of the clock used in timing the functions is too large to provide an accurate picture of how processing time is distributed amongst the respective functions. Instruction counting is another statistical technique for monitoring performance of a program. Instruction counting techniques have generally been limited to monitoring the performance of user level programs (as opposed to system level programs), such as application programs. These techniques have not been applicable to monitoring the performance of operating system kernels. As a result of these limitations, statistical techniques have often not been useful to a programmer to help him enhance program performance.

Summary of the Invention

A method is practiced in a data processing system having a processor for executing instructions and a storage mechanism for storing an operating system and a user level monitoring program. The processor executes instructions at a user level and a system level. The data processing system may be a distributed system. In this method, a facility is provided in the kernel of the operating system to monitor program performance. For instance, the facility may count the instructions executed by the processor, count the number of times that a section of code has been called during execution of a program or provide a number of separate counts categorized by function that specify the number of instructions executed in the respective functions. A section of code of a program is executed on the processor. The facility in the kernel of the operating system is then used to monitor system performance. The type of facility used determines the values that are counted by the facility. The counts are reported to the user level monitoring program. For example, the facility may count the number of instructions when executing a section of code of the program or may count the number of times that a section of code of the program is called or the facility may provide separate counts categorized by function with instructions executed in the functions when a first function is called during execution of the program. The program may be a user level program or part of the kernel of the operating system.

Brief Description of the Drawings

Figure 1 is a block diagram of a distributed system for practicing a preferred embodiment of the present invention.

Figure 2 is a block diagram showing a computer system of Figure 1 in more detail.

Figure 3 is a block diagram showing an API of Figure 2 in more detail.

Figure 4 is a flowchart illustrating the steps performed from a user's perspective in monitoring program performance and tuning the program in accordance with the preferred embodiment of the present invention.

Figure 5 is a flowchart illustrating in more detail how breakpoints are inserted and how a program is run when monitored by the preferred embodiment of the present invention.

Detailed Description of the Invention

A preferred embodiment of the present invention provides a facility within the kernel of an operating system to monitor performance of programs. The facility provides a user with a number of options. First, The facility allows a user to count the number of instructions executed by a function each time that the function is called. Second, the facility allows the user to count the number and frequency of calls to a particular function without counting the instructions executed in the function, and third, the facility allows the user to count the number of instructions executed in each function that is called as a result of calling an initial function. Since the facility is provided in the kernel of the operating system, it may be used not only to monitor the performance of user level programs, such as application programs, but also may be used to monitor the performance of the kernel and other portions of the operating system. Providing the facility within the kernel of the operating system also enhances the speed of monitoring and provides a nonintrusive approach to performance monitoring that does not require any modifications to the program being monitored.

The preferred embodiment of the present invention may be used in a distributed system 10 like that shown in Figure 1. It should also be appreciated that the present invention may be practiced in single processor system. The distributed system includes a number of computer sys-

tems 12 that communicate through a network 14. Each of the computer systems 10 may concurrently run a trace on the program it is executing. The network may be any of a number of different types of networks, including local area networks (LANs) or wide area networks. Those skilled in the art will appreciate that the distributed system 10 may include a different number of computer systems than shown in Figure 1. The depiction of four computer systems in the distributed system is merely illustrative.

Figure 2 is a more detailed block diagram of the components contained in one of the computer systems 12. Each of the computer systems need not have this configuration, but this configuration is intended to be merely illustrative. The computer system 12 includes a central processing unit (CPU) 13 and a memory 16. The CPU has a number of registers 17, including a FLAGS register, and a 55 (Stack Segment) register. These registers will be described in more detail below. The memory 16 holds a copy of a distributed operating system 28 such as the Microsoft NT Operating System, sold by Microsoft Corporation of Redmond, Washington. In the preferred embodiment of the present invention, each of the computer systems 12 has a copy of the operating system that it runs. The operating system 28 includes a kernel 30, which, in turn, includes a section of code 32 that provides support for performance monitoring. The section of code 32 will be described in more detail below. The memory 16 may also hold at least one application program 26 that may be run on the CPU 13. The application program 26, however, need not reside in memory 16.

The computer system 12 also includes a number of peripheral devices. These peripheral devices include a secondary storage device 18, such as a magnetic disc drive, a keyboard 20, a video display 22 and a mouse 24.

Figure 3 is a block diagram showing the section of code 32 in more detail. The API includes a breakpoint facility 34 for supporting the generation of breakpoints within a program run on the CPU 13. The section of code 32 also includes a single step interrupt handler 36 for handling single step interrupts. In order to explain the role served by the breakpoint facility 34 and the single step interrupt handler 36, it is helpful first to review what breakpoints are and what single step interrupts are.

A single step interrupt is an interrupt that is used when a program is single-stepping. Single-stepping refers to a mode of operation of the CPU 13 in which one instruction is executed at a time under the control of a supervisory program. Single-stepping provides a mechanism for debuggers and performance monitors to execute a program slowly to carefully monitor execution of the program. In the present instance, the single step handler 36 is the supervisory program under which a program being monitored executes one instruction at a time. In general, an instruction is executed and then a single step interrupt is generated to switch control of the processor to the supervisory program. Many microprocessors, such as the 80X86 microprocessors, provide a convenient mechanism for switching the microprocessor into single-stepping mode. For the 80X86 microprocessors, a single step trap flag is provided in the FLAGS register (see Figure 2). When the single step trap flag has a value of one, the processor executes in single step mode and, conversely, when the single step trap flag has a value of zero, the microprocessor does not operate in single-stepping mode.

A breakpoint refers to a breakpoint interrupt that is used to break the uninhibited execution of a program and is helpful in debugging and performance monitoring programs. The difficulty with executing an entire program in single-stepping mode is that it is very slow. In many instances, the majority of a program may already be debugged or tuned and only a small portion needs to be examined in more detail. A breakpoint interrupt is especially suitable in such circumstances. A breakpoint interrupt is generated by inserting a special opcode into a program. This opcode, when executed, causes a breakpoint interrupt to be generated. Upon generation of the breakpoint interrupt, control is transferred to the breakpoint facility 34, which may then perform further actions as needed. The role of breakpoints and single-step interrupts in the preferred embodiment of the present invention will be described in more detail below with regard to the discussion of Figures 4 and 5.

In addition to the breakpoint facility 34 and the single step handler 36, the section of code 32 also includes code relating to context switching 38. This code 38 provides a hook for context switches and includes a handler that looks at the context switch and determines whether the new context should be single-stepped or not. If the new context is to be single-stepped, the handler makes certain that single-stepping occurs.

The section of code 32 also includes counter values 40. As was discussed above, the preferred embodiment of the present invention provides a user with an opportunity to monitor the number of instructions executed in the function, the number of times a function is called within a program and the number of instructions executed for each function that is called as a result of calling an initial function. Each of these counts is stored in a separate counter, and the values of the counters are held as counter values 40 in the data area of the section of code 32.

Figure 4 is a flow chart of the steps performed to monitor the performance of a program or a portion of a program within the preferred embodiment of the present invention. Initially, the user selects a performance metric (step 42). A user interface, such as a menu of dialog box, is provided on the video display 22 (Figure 2) of the computer system 12 (Figure 1) that he is using, to enable the user to select the desired performance metric. As was discussed above, there are a number of different performance metrics that may be calculated by the preferred embodiment of the present invention. The user also chooses what functions in the program are to be monitored (step 44) . Once the performance metrics have been selected and the functions to be monitored have been selected, opcodes for generating breakpoint interrupts are inserted into each of the functions that is to be monitored, and the program or portion of the program is executed (step 46). The appropriate counter values 40 are calculated as the program is run. The counter values 40 are displayed on the video display 22 as the program executes (step 48 in Figure 4). When the program or portion of a program is done executing, the user may examine the performance data to guide him in appropriately tuning the program (step 50) . How the data is used to tune the program depends upon the type of data that is gathered. Those skilled in the art will appreciate how to use the performance metrics data to appropriately tune the program.

Figure 5 is a flow chart showing the steps performed on a function-by-function basis in inserting a breakpoint and executing the program (i.e., step 46 in Figure 4) in more detail. As mentioned above, a breakpoint opcode is placed at the first instruction of each function to be monitored (step 52 in Figure 5). The system then begins execution of the function (step 54). Since the breakpoint is the first instruction in the function, the kernel catches the breakpoint (step 56) and then determines what performance metric the user has requested for the function (step 58). If the only performance metric requested is a count of the number of calls to the function, a counter in the counter values 40 (Figure 3), that tracks the number of calls to the function, is incremented to indicate that the function has been called (step 60). Once the counter has been incremented, the function is restarted and resumes execution at full speed.

Before considering the steps that are performed in the preferred embodiment of the present invention when the user wishes to monitor the number of instructions executed, it is first helpful to introduce the notion of a "thread" and a "stack." A thread represents an executable portion of a task that can be run independently and concurrently with other threads. The operating system 28 is a multithreading system. Those skilled in the art will appreciate that the present invention need not be practiced in a multithreaded system. It may also be practiced in single process and multiprocessing systems. A multithreading operating system is able to execute multiple threads concurrently. The stack is a last-in-first-out (LIFO) data structure that is used to hold data and register information for the thread. Each thread has a portion of the stack segment dedicated to it. The SS (Stack Segment) register (Figure 2) points to a segment in memory that holds stacks for the threads of the program being executed. Each thread has a stack pointer (SP) value associated with it that points to the top of its stack (i.e., the portion of the stack segment dedicated to the thread). Those skilled in the art will appreciate that the present invention may also be practiced in environments that do not use stacks.

When in step 58 of Figure 5 it is determined that the user wishes to count a number of instructions executed rather than counting the number of function calls, the stack pointer for the currently executing thread is recorded in memory 16 (step 64). The single step trap flag in the FLAGS register (Figure 2) is then set to cause the CPU 13 to enter single-stepping mode (step 66). In single-stepping mode, each time that an instruction is executed, the counter value that keeps track of the number of instructions executed for the current function is incremented (step 68). The instructions are counted until the thread exits the function being monitored or the thread terminates. The instruction counting may also be suspended when a thread deschedules itself, but the instruction counting resumes after the thread is again scheduled.

While the present invention has been described with reference to a preferred embodiment thereof, those skilled in the art will appreciate that various changes in form and detail may be made without departing from the present invention as defined in the appended claims.

Claims

1. In a data processing system having a processor that runs at a system level or a user level when executing instructions and a storage mechanism for storing an operating system that includes a kernel, a method comprising the steps of:

   providing a facility in the kernel of the operating system for counting instructions executed by the processor and a user level monitoring program for monitoring the instructions counted by the facility;

   executing a section of code on the processor;

using the facility in the kernel of the operating system to count the number of instructions executed when executing the section of code on the processor; and

reporting the count of the number of instructions executed to the user level monitoring program.

2. The method recited in claim 1 wherein the section of code that is executed on the processor is part of the kernel of the operating system.

3. The method recited in claim 1 wherein the section of code that is executed on the processor is part of a user level program.

4. The method recited in claim 1 wherein the section of code that is executed on the processor is a non-kernel portion of the operating system.

5. The method recited in claim 1, further comprising the step of storing the count of the number of instructions executed when executing the section of code on the processor in the storage mechanism.

6. The method recited in claim 1 wherein the section of code is a function and the step of executing the section of code on the processor further comprises the step of executing the function on the processor.

7. The method recited in claim 1, further comprising the step of tuning the section of code executed on the processor to decrease the number of instructions that are executed when the section of code is executed on the processor.

8. The method recited in claim 1 wherein the data processing system is a distributed system that also includes a second processor and wherein the method further comprises the steps of:

executing a second section of code on the second processor;

using the facility in the kernel of the operating system to count the number of instructions executed when executing the second section of code on the second processor; and

reporting the count of instructions executed when executing the second section of code to the user level monitoring program.

9. The method recited in claim 8, further comprising the step of storing the count of the number of instructions executed when executing the second section of code on the second processor in the storage mechanism.

10. In a data processing system having a processor that runs at a system level or a user level when executing instructions and a storage mechanism for storing an operating system that includes a kernel, a method comprising the steps of:

providing a facility in the kernel of the operating system for counting a number of times that a section of code has been called during execution of a program and a user level monitoring program for monitoring the instructions counted by the facility;

executing at least a portion of a program that includes a first section of code on the processor;

using the facility in the kernel of the operating system to count a number of times that the first section of code is called during execution of the portion of the program on the processor; and

reporting the count to the user level monitoring program.

11. The method recited in claim 10 wherein the program is part of the kernel of the operating system.

12. The method recited in claim 10 wherein the program is a user level program.

13. The method recited in claim 10 wherein the program is a non-kernel portion of the operating system.

14. The method recited in claim 10, further comprising the step of storing the count of the number of times that the first section of code is called during execution of the portion of the program in the storage mechanism.

15. The method recited in claim 10 wherein the first section of code is a function and the step of using the facility in the kernel of the operating system to count the number of times that the first section of code is called during execution of the portion of the program further comprises the step of using the facility in the kernel of the operating system to count the number of times that the function is called during execution of the portion of the program.

16. The method recited in claim 10, further comprising the step of using the count of the number of times that the first section of code

is called during execution of the portion of the program to direct tuning of performance of the program.

17. The method recited in claim 10 wherein the program includes a second section of code and wherein the method further comprises the step of using the facility in the kernel of the operating system to count a number of times that the second section of code is called during execution of the portion of the program on the processor and reporting the count of times that the second section of code is called to the user level monitoring program.

18. The method recited in claim 10, further comprising the step of storing the count of the number of items that the second section of code is called during execution of the portion of the program on the processor in the storage mechanism.

19. The method recited in claim 10 wherein the data processing system is a distributed system that also includes a second processor and wherein the method further comprises the steps of:

executing at least a portion of a second program that includes a section of code on the processor;

using the facility in the kernel of the operating system to count a number of times that the section of code of the second program is called during execution of the portion of the second program; and

reporting the count of the number of times that the section of code of the second program is called to the user level monitoring program.

20. The method recited in claim 19, further comprising the step of storing the count of the number of times that the section of code of the second program is called during the execution of the portion of the second program in the storage mechanism.

21. In a data processing system having a processor capable of running at a system level and a user level for executing instructions and a storage mechanism for an operating system having a kernel, a method comprising the steps of:

providing a facility in the kernel of the operating system for counting instructions executed by the processor;

executing a first function that calls other functions in the program on the processor;

using the facility in the kernel of the operating system to provide separate counts categorized by function of instructions executed in the first function and the other functions when the first function is called during execution of the program; and

reporting the counts to the user level monitoring program.

22. The method recited in claim 21 wherein the program is part of the kernel of the operating system.

23. The method recited in claim 21 wherein the program is a user level program.

24. The method recited in claim 21 wherein the program is a non-kernel portion of the operating system.

25. The method recited in claim 21, further comprising the step of storing the separate counts in the storage mechanism.

26. The method recited in claim 21, further comprising the step of tuning performance of the program using the separate counts to direct the tuning.

27. A system for measuring performance of a monitored program of instructions, comprising:

a storage mechanism for storing a user level monitoring program and an operating system that includes a kernel, said kernel comprising a facility for counting instructions that are executed; and

a processor comprising:

(i) a program execution unit for executing the monitored program and the user level monitoring program;

(ii) a facility invoker for invoking the facility to count the number of instructions executed by at least a portion of the monitored program; and

(iii) a count reporter for reporting the counts gathered by the facility to the user level monitoring program.

28. A system for measuring performance of a monitored program of instructions, said monitored program including a function, the system comprising;

a storage mechanism for storing a user level monitoring program and an operating system that includes a kernel, said kernel comprising a facility for counting a number of times that a function is called;

a processor comprising:

(i) a program execution unit for executing the monitored program and the user level

monitoring program;
(ii) a facility invoker for invoking the facility to count the number of times that the function in the monitored program is called during execution of the monitored program; and
(iii) a count reporter for reporting the count to the user level monitoring program.

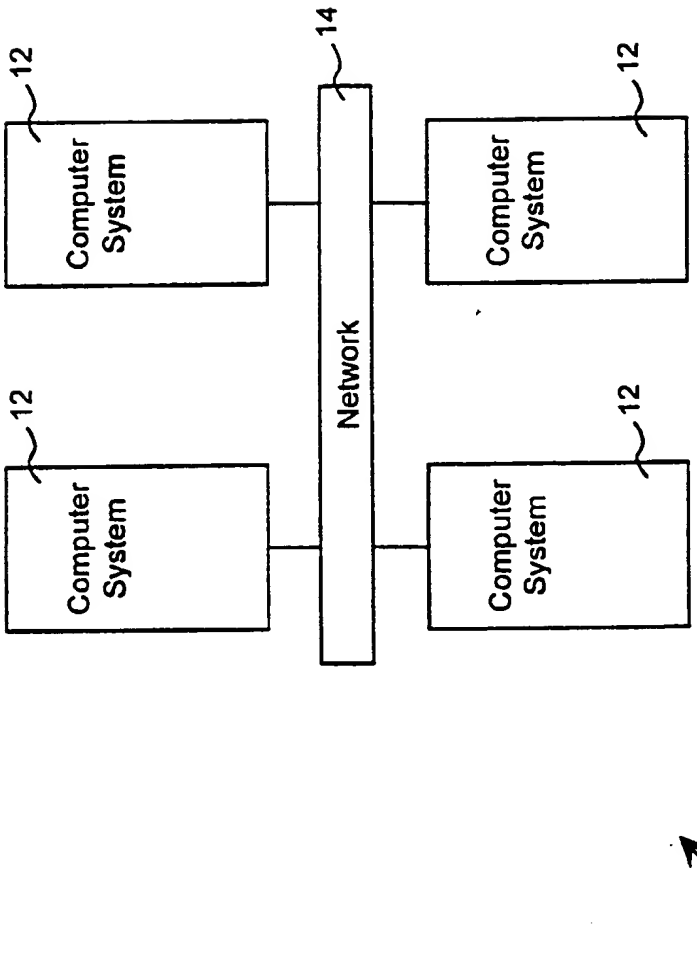
12
Computer System
14
Network
12
Computer System
12
Computer System
12
Computer System
10

Figure 1

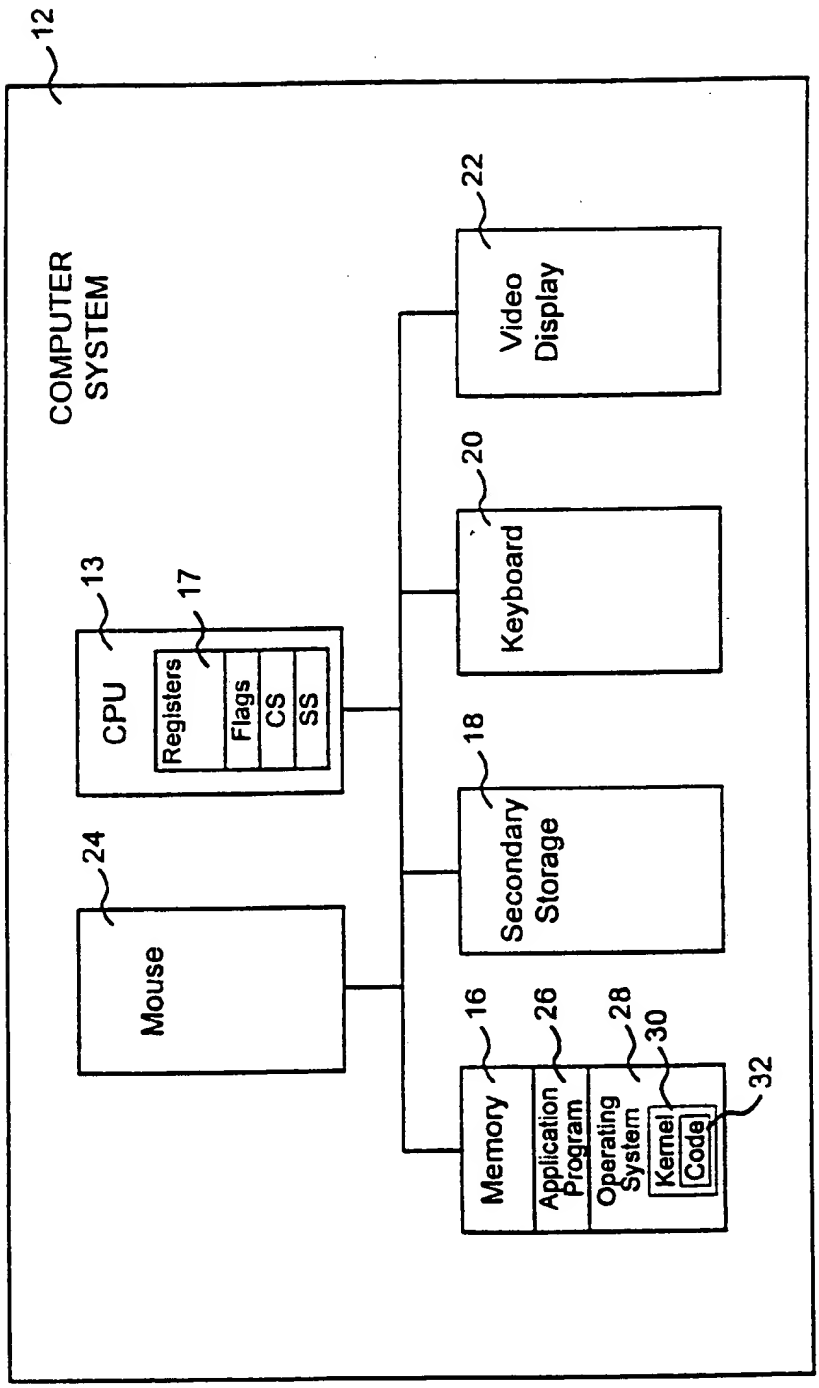
12
COMPUTER SYSTEM
CPU
13
Registers
17
Flags
CS
SS
Mouse
24
Secondary Storage
18
Keyboard
20
Video Display
22
Memory
16
Application Program
26
Operating System
28
Kernel
30
Code
32

Figure 2

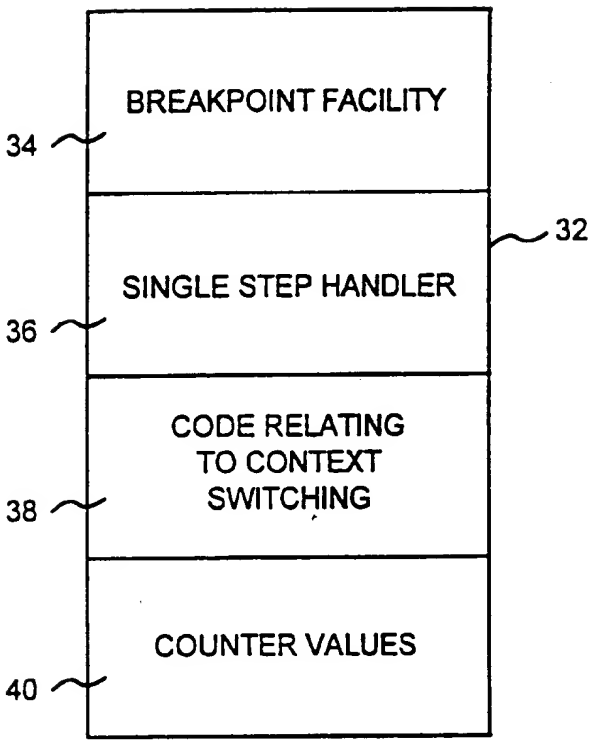
BREAKPOINT FACILITY
34
SINGLE STEP HANDLER
36
32
CODE RELATING TO CONTEXT SWITCHING
38
COUNTER VALUES
40

Figure 3

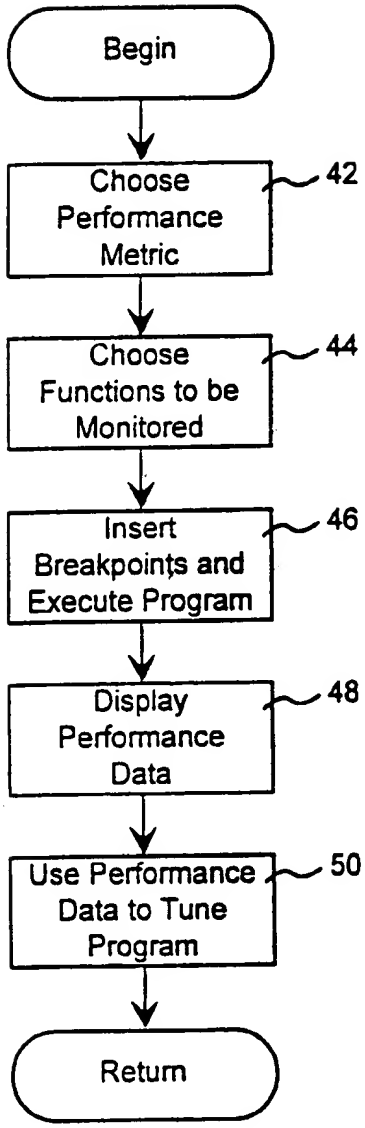
Begin
Choose Performance Metric
42
Choose Functions to be Monitored
44
Insert Breakpoints and Execute Program
46
Display Performance Data
48
Use Performance Data to Tune Program
50
Return

Figure 4

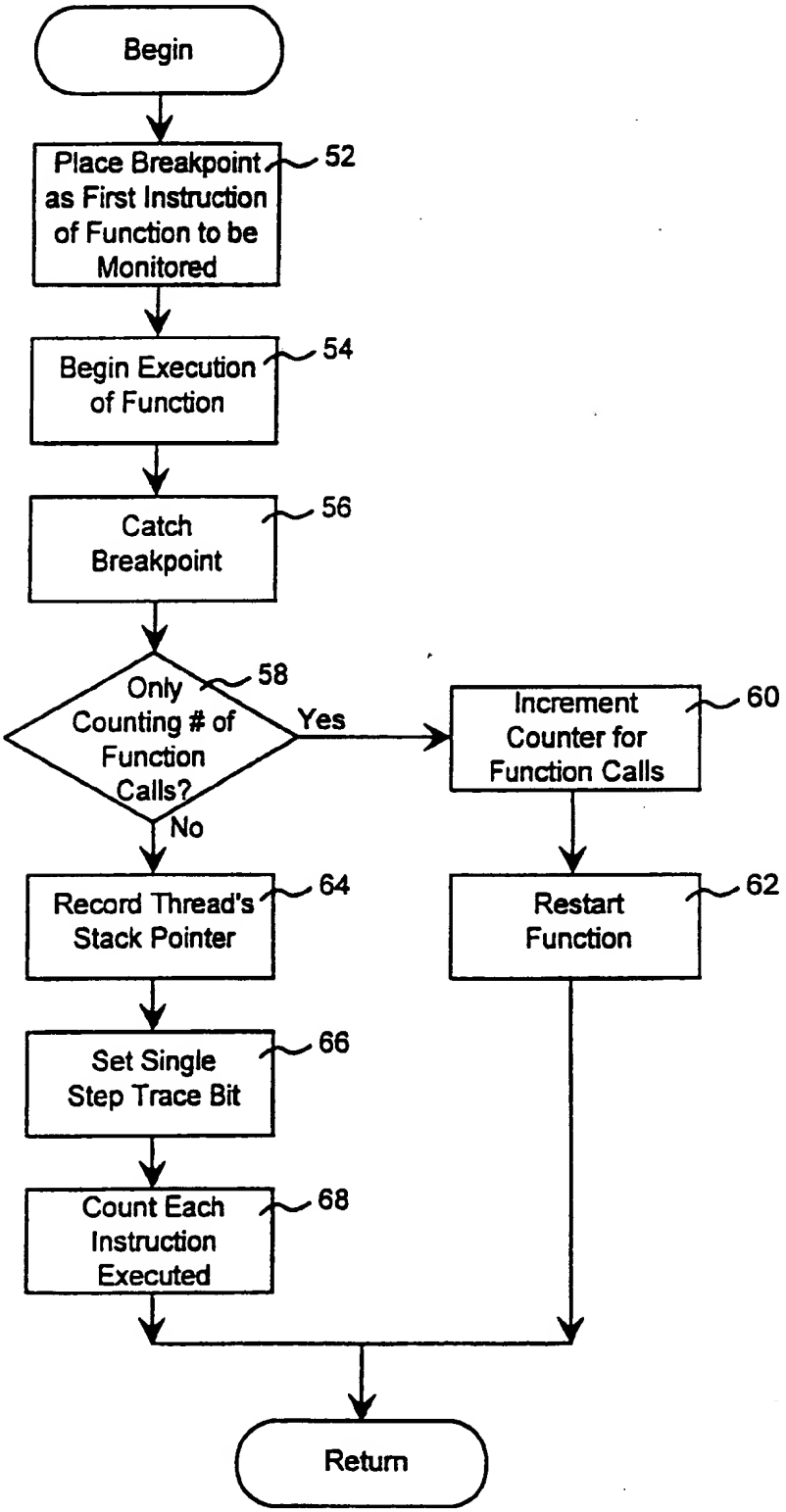
Begin
Place Breakpoint as First Instruction of Function to be Monitored
52
Begin Execution of Function
54
Catch Breakpoint
56
Only Counting # of Function Calls?
58
Yes
No
Increment Counter for Function Calls
60
Restart Function
62
Record Thread's Stack Pointer
64
Set Single Step Trace Bit
66
Count Each Instruction Executed
68
Return

Figure 5

European Patent Office

EUROPEAN SEARCH REPORT

Application Number
EP 94 11 7366

DOCUMENTS CONSIDERED TO BE RELEVANT

Category	Citation of document with indication, where appropriate, of relevant passages	Relevant to claim	CLASSIFICATION OF THE APPLICATION (Int.Cl.6)
X	EP-A-0 526 055 (RESEARCH MACHINES PLC) * column 5, line 51 - column 6, line 18 * * column 7, line 38 - column 8, line 4 * * column 9, line 28 - line 36 *	1-28	G06F11/34
A	IBM TECHNICAL DISCLOSURE BULLETIN, vol.36, no.7, July 1993, NEW YORK US pages 39 - 42 'Translation of Data Generated by the AIX 3.2 Trace Facility into a Format for Visualization of the Data' * page 40, line 2 - line 6 *	2-4, 11-13, 22-24	
A	IBM SYSTEMS JOURNAL, vol.22, no.3, 1983, ARMONK, NEW YORK US pages 271 - 294 L. R. POWER 'Design and use of a program execution analyzer' * page 284, right column, line 18 - line 27; figure 5 *	1-28	
			TECHNICAL FIELDS SEARCHED (Int.Cl.6) G06F

The present search report has been drawn up for all claims

Place of search	Date of completion of the search	Examiner
THE HAGUE	21 February 1995	Corremans, G

CATEGORY OF CITED DOCUMENTS

X : particularly relevant if taken alone
Y : particularly relevant if combined with another document of the same category
A : technological background
O : non-written disclosure
P : intermediate document

T : theory or principle underlying the invention
E : earlier patent document, but published on, or after the filing date
D : document cited in the application
L : document cited for other reasons

& : member of the same patent family, corresponding document

EPO FORM 1503 03.82 (P04C01)